MODELLBASIERTE GENERIERUNG VON SPRACHDIALOGEN FÜR EINGEBETTETE SYSTEME

Uwe Koloska, Matthias Pohl

voice INTER connect GmbH
koloska@voiceinterconnect.de

Kurzfassung Im Bereich der Spracherkennung für eingebettete Systeme wird die Interaktion zwischen Mensch und Maschine vielfach durch vorgegebene mehr oder weniger flexible Dialoge in feste Bahnen gelenkt, um dem Spracherkennungssystem die zusätzlichen semantischen Informationen zu verschaffen, die es für eine sichere Erkennung der relevanten Inhalte benötigt. Solche Dialogsysteme werden bisher in fast allen Fällen von Hand direkt für die Zielplattform kodiert und lassen sich nur in Ausnahmen wiederverwenden. Durch eine formale Beschreibung der Dialog- und der Zielplattform mit Hilfe von domänenspezifischen Sprachen (DSL) kann durch modellbasierte Programmierung ein abstrakter Dialog in eine Implementierung der konkreten Zielplattform transformiert werden.

Der Vortrag beschreibt ein Projekt, bei dem ein Editor für Sprachdialoge mit Hilfe von modellbasierten Prinzipien auf Basis der *Eclipse Modeling Platform* entwickelt wurde und präsentiert einen Ansatz zur Erzeugung der notwendigen Metamodelle für die Dialog- und die Zielplattform mithilfe einer Mustersprache (Pattern Language).

1 Sprachdialogentwicklung und Anforderungen

Heute kommt Sprachbedienung vor allem in mobilen und miniaturisierten Anwendungen zum Einsatz, wo der Anwender die Hände nicht nutzen kann oder der visuelle Kanal belegt ist (Automobil) oder aufgrund von Miniaturisierung kein Platz für Bedienelemente zur Verfügung steht. Sprachbedienung wird am besten so mit herkömmlichen Bedienkonzepten (haptisch oder GUI-basiert) integriert, dass eine intuitive Nutzung erfolgen kann und sich ein selbsterklärendes und geschlossenes Bedienkonzept ergibt. Eine Herausforderung bei der Umsetzung eines multimodalen Bedienkonzeptes ist die Sicherstellung von Echtzeitfähigkeit trotz begrenzter Ressourcen in einem Embedded System, die Implementierung von Systemintelligenz und durch sorgfältige Auswahl von Steuerbefehlen, eine akustische Rückkopplung durch Sprachausgabe, sowie enge Synchronisation zwischen Sprachbedienung, GUI, haptischer Bedienung und Applikationszustand.

Sprachbedien- und Sprachausgabefunktionen werden dabei in Form eines sogenannten Sprachdialoges präsentiert, welcher Zustände, eine Ereignisverarbeitung und Zustandsübergänge aufweist. Für ein positives Bedienerlebnis darf der Nutzer niemals das Gefühl erhalten, in seinen Möglichkeiten eingeschränkt zu sein. Ein solches Bedienerlebnis kann – ebenso wie gutes Design – die Kaufentscheidung des Nutzers stark beeinflussen.

Für die Entwicklung einer funktionsfähigen Sprachbedienung ergeben sich aus den genannten Zielen folgende Forderungen:

- einheitlicher Zugriff auf eine Vielzahl von Ressourcen für Spracherkennung und Sprachausgabe (sprachspezifische Wissensbasen, Lexika, Regelwerke, Sprachprompts etc.)

- Beschreibung und Verwaltung der Sprachdialogstruktur sprachenunabhängig
- Visualisierungsmöglichkeiten der komplexen Dialogstruktur (z. B. in Form einer grafischen Repräsentation)
- leistungsfähige und dennoch einfach zu bedienende Entwurfsumgebung zur Benutzung durch Sprachdialogdesigner (die u. U. keine geübten Programmierer sind)
- mit Sprachdialog synchronisierter Entwurf der GUI durch Grafikdesigner (z. B. Beschriftung der Bedienelemente, Zustandswechsel, Kombination sprachlicher und grafischer Bedienung)
- umfangreiche Simulations- und Testmöglichkeiten unabhängig vom Zielgerät
- effektive Umsetzung der Dialogbeschreibung auf der Zielplattform durch Programmierer

Eine Implementierung des Sprachdialoges wird häufig – ebenso wie die GUI – als Zustandsautomat realisiert. Die Berücksichtigung von Systemereignissen (aus Sprachbedienung und Bedienhandlungen sowie Ereignissen aus der Applikation) erlaubt eine enge Verknüpfung aller Systemkomponenten. Eine effektive Implementierung und Synchronisation aller Bedienmodi und der Applikation trägt zu einer hohen Nutzerakzeptanz des resultierenden Systems bei.

2 Modellgetriebene SW-Entwicklung

Da es typische Aufgaben für Sprachbedienung gibt (z.B. Telefon- und Gerätebedienung, Zieleingabe beim Navigationssystem, Zugriff auf Medienbibliotheken etc.), die auf unterschiedlichen Geräten identisch ausgeführt werden, ist es sinnvoll, bereits erstellte Sprachdialoge wiederzuverwenden und nach kleineren kunden- oder produktspezifischen Anpassungsmaßnahmen auf einer anderen Zielplattform umzusetzen.

Eine für die Entwicklung von Sprachdialogen geeignete Lösung stellt die modellgetriebene Softwareentwicklung (MDSD, *Model Driven Software Development*[10]) dar. Dabei wird ein Modell des Dialogs erstellt, das soweit wie möglich von einer konkreten Zielplattform abstrahiert[1], und daraus der konkrete Programmcode der Zielapplikation erzeugt.

Die grundlegenden Prinzipien der modellbasierte Entwicklung können in den Antworten auf zwei Fragen zusammengefasst werden [1]:

> What is a model? *A model is a simplification of reality.*
>
> Why do we model? *We build models so that we can better understand the system we are developing.*

Vorteile des modellbasierten Ansatzes bestehen vor allem darin, dass die Dialogerstellung streng von der Implementierung auf einer bestimmten Zielplattform getrennt wird. Der Dialogdesigner kann sich ganz auf die Gestaltung des Dialogs konzentrieren und der Programmierer den optimalen Code zur Abbildung des Dialogmodells erarbeiten, der für alle möglichen Ausprägungen des Dialog-Metamodells wiederverwendet werden kann.

[1] Eine vollständige Abstraktion hätte bei eingebetteten Systemen immer einen Einfluss auf die Performance des erzeugten Programmcodes oder auf die Komplexität des Codegenerators, da entweder Sonderfälle nicht abgedeckt wären oder mit großem Aufwand aus dem abstrakten Modell des Dialogs extrahiert werden müssten.

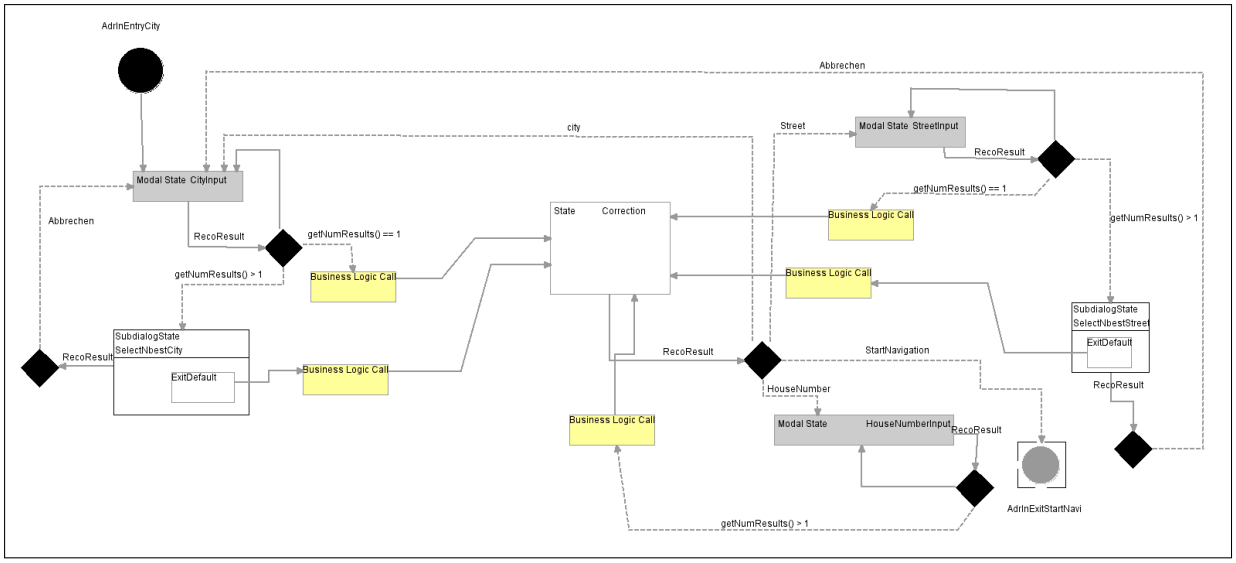

Abbildung 1 - Modell eines Dialogs zur Adresseingabe

Ein template- oder generatorbasiertes Verfahren (M2T, Model to Text Transformation) zur Implementierung auf der Grundlage einer plattformunabhängigen Dialogbeschreibung (Dialogmodell, siehe Abb. 1 für das Beispiel eines solchen Dialogmodells) effektiviert den Entwurfsprozess, minimiert manuellen Aufwand und verbessert somit die Qualität und die Testbarkeit des Codes. Durch eine zusätzliche grafische Dialogrepräsentation wird begleitend zum Entwurfsprozess die Dokumentation der Leistung erstellt. Für die Umsetzung eines solchen Verfahrens auf der Zielplattform gibt es zwei Alternativen:

1. Ein mächtiges Framework, das zur Umsetzung des Dialogs auf der Zielplattform benutzt wird und zusammen mit der Zielanwendung compiliert wird.

2. Ein universelles Dialogprogramm für die jeweilige Zielplattform, das den eigentlichen (plattformunabhängigen) Dialog aus einer Konfigurationsdatei liest und dann ausführt. (Interpreter-Lösung)

Für eingebettete Systeme haben diese Ansätze aber den Nachteil, dass sie um nützlich zu sein, immer auch nahezu vollständig sein müssen und damit bei einer durchschnittlichen Dialogimplementierung sehr viel Overhead produzieren.[2]

Durch Anwendung der modellbasierten Entwicklung können beide Verfahren ihre ganze Stärke ausspielen, da bei fertigem und optimiertem System aus Generator und Zielplattform, jeder modellierte Dialog für dieses Ziel erzeugt werden kann. Laufzeitoptimierungen und spezielle Anpassungen an eine bestimmte Umgebung können damit sofort von allen Dialogen genutzt werden, ohne dass manuelle Anpassungen vorgenommen werden müssen. Und die Komplexität des Frameworks, als auch des Interpreters könnte über generierte Komponenten auf das Nötigste reduziert werden: Wenn der Dialog ein bestimmtes Feature nicht benötigt (was der Generator direkt aus dem Modell abfragen kann), kann dieses aus dem Framework entfernt werden.

3 Eclipse Modeling Project

Im Rahmen des Eclipse Modeling Projects (siehe [8, 6]) stehen praxiserprobte Komponenten bereit, die mit Hilfe der modellbasierten Entwicklung auf Grundlage des Eclipse Modeling

[2]Dieser Nachteil ließe sich natürlich durch einen Compiler für die Dialogsprache umgehen.

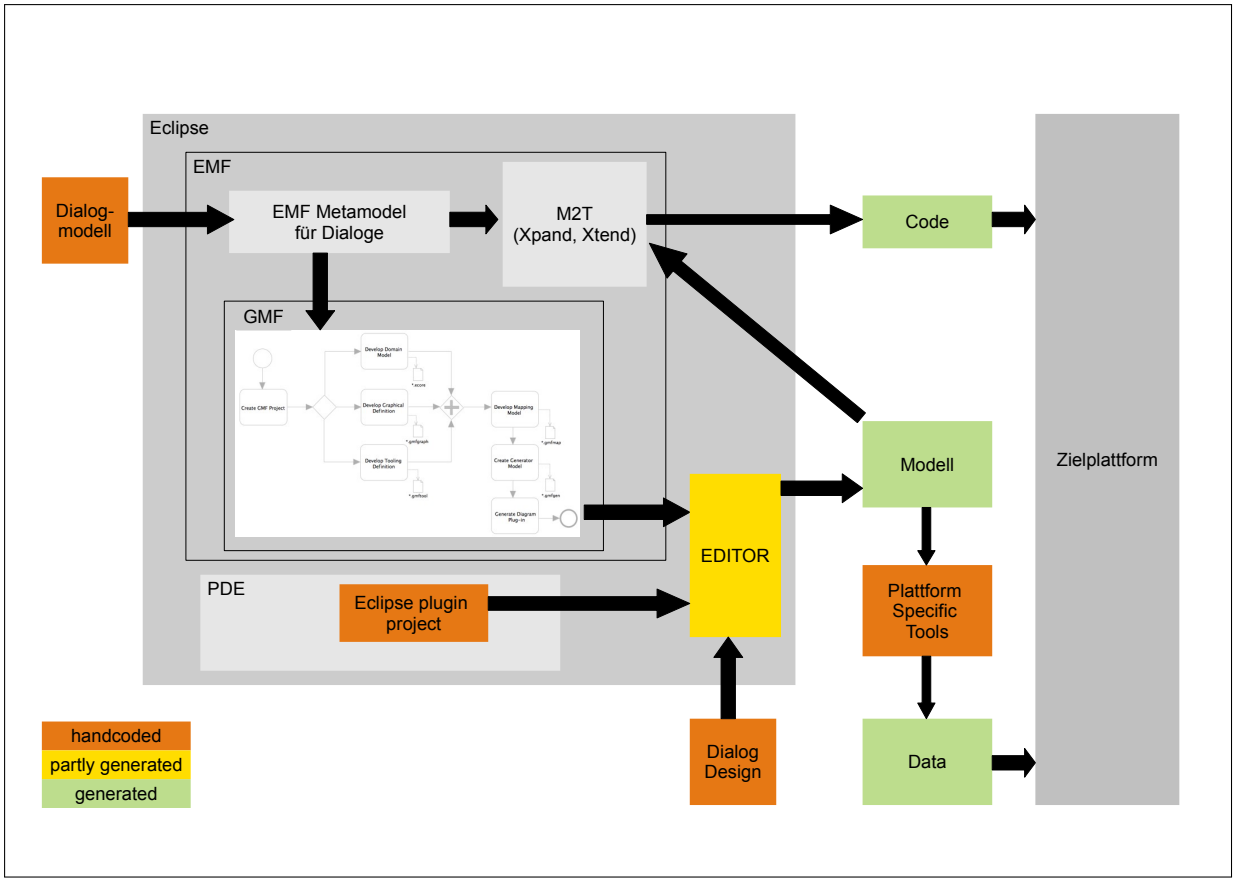

Abbildung 2 - Übersicht der Entwicklung im Eclipse Modeling Project

Frameworks (EMF [2]) einen grafischen Editor basierend auf einem Metamodell der DSL erzeugen können. Zusammen mit dem *openArchtitectureWare* Projekt (oAW [9])[3], das die Bausteine für den Codegenerator und die Validierung des Modells lieferte, waren die grundlegenden Bestandteile für das gewünschte System vorhanden.

Abbildung 2 zeigt einen Überblick über die verwendeten Komponenten und ihr Zusammenspiel, um aus dem Metamodell des Dialogs einen grafischen Editor und einen Codegenerator für die Zielplattform zu bauen. Ein Vorteil der Nutzung von Eclipse Komponenten ist, dass alle für die modellbasierte Entwicklung nötigen Module vorhanden und in ein gemeinsames Applikationsframework (RCP, Rich Client Platform) integriert sind. Der größte Nachteil ist, dass die Lernkurve für die Plattform und die Integration der einzelnen Komponenten sehr steil und die vorhandene Dokumentation nicht immer vollständig oder auf dem aktuellen Stand ist. Der größte Vorteil ist allerdings, dass alle Komponenten unter der *Eclipse Public License* stehen, die eine Verwendung auch im kommerziellen Kontext erlaubt (siehe [5]).

4 Praxisbeispiel

Für einen Kunden sollte ein grafischer Editor für Sprachdialoge erstellt werden, der es einem Dialogdesigner (der normalerweise kein Entwickler für die Zielplattform ist) erlaubt, einen Dialog zu erstellen und im Ablauf sowohl mit als auch ohne Zugriff auf die Zielhardware zu testen. Gleichzeitig sollte die Möglichkeit bestehen, die einmal erstellten Dialoge auch für andere Zielplattformen in lauffähige Applikationen umzuwandeln. Die zur Verfügung stehende Zeit, machte es unmöglich, die Vorgaben durch eine Neuentwicklung zu erfüllen, und so musste auf

[3] oAW ist mittlerweile (seit Eclipse 3.5 Galileo) zu großen Teilen Bestandteil des Eclipse Projekts

vorhandene Komponenten zurückgegriffen werden.

Mit Hilfe der Komponenten des Eclipse Frameworks konnte in der vorgesehenen Zeit ein grafischer Editor für Dialoge mit Validierung und Codegenerierung erstellt werden (siehe Abb. 3), der die Entwicklung und das Testen von Dialogen auch für Dialogdesiger ermöglicht.

5 Praktische Erfahrungen

Im Laufe der Arbeit mit modellbasierten Prinzipien im Allgemeinen und der konkreten Implementierung unter Eclipse wurden die im folgenden dargestellten Erfahrungen gesammelt, welche in Folgeprojekten berücksichtigt werden.

5.1 Fokus auf das Metamodell

Die wichtigste Lektion war: Im Zentrum der ganzen Entwicklung steht das Metamodell des Dialogs, denn damit steht und fällt die Umsetzung der Dialogbeschreibung auf die Zielplattform und die Bedienung des Dialogtools. Durch die verwendeten Frameworks bestimmt das Metamodell den Aufbau der erzeugten Programme.

5.2 Trennung zwischen Dialogmodell und Zielplattform

Das Metamodell sollte, wie weiter oben schon angesprochen, möglichst wenig Informationen über eine konkrete Zielplattform enthalten, sonst ist die Allgemeingültigkeit des Modells beschränkt und ein Plattformwechsel gestaltet sich schwierig. Im Bereich der eingebetteten Systeme müssen einige Elemente der Zielplatform(en) auch im Dialogmodell darstellbar sein, aber diese sollten – wenn irgend möglich – von der konkreten Implementierung abstrahiert werden.

Ein verbreiteter Fehler beim Formulieren des Metamodells ist, dafür schon in der Diktion einer Statemachine zu denken (meist sogar in der konkreten Implementierung für eine Zielplattform oder zumindest der Vorgaben der UML2). Dadurch verschenkt man aber die Möglichkeit, einen Dialog adäquat zu modellieren und denkt nur noch in den Grenzen des bekannten Metamodells.

5.3 Modell für die Zielplattform

Ein explizites Modell für die Zielplattform ermöglicht eine bessere Trennung der einzelnen Modelle und der auf ihnen durchgeführten Operationen wie der Codegenerierung. Dieses dient als Ziel für eine Modelltransformation des Dialogmodells entlastet die Codegenerierung von Funktionen, die aus den Informationen des Dialogmodells Eigenschaften der Zielplattform ableiten.

Da das verwendete Dialogmodell nicht sehr weit von der Darstellung des Dialogs auf der Zielplattform abstrahiert, setzt die Codegenerierung direkt auf dem Dialogmodell selbst auf.

Ein Beispiel aus dem bearbeiteten Projekt: Ein Dialog kann im Editor in mehreren Applikationen benutzt werden. Für die Zielplattform gehören aber Applikation und Dialog immer zusammen. So muss bei der Codegenerierung der von einer Applikation unabhängige Dialog mit dieser zusammengeführt werden. Dadurch wird die Codegenerierung unnötig kompliziert

5.4 Modellelemente nur für die Darstellung

Leider haben es die verwendeten Komponenten (besonders das GMF, Graphical Modeling Framework) nötig gemacht, dass einige Elemente des Dialogmodells in Subklassen aufgesplittet werden mussten, um eine getrennte Konfiguration der Darstellung zu ermöglichen. Dadurch

wurde das Metamodell des Dialogs deutlich umfangreicher und komplizierter als nötig. Hier musste pragmatisch von der üblichen Vorgehensweise (siehe [7]) abgewichen werden.

5.5 Aufteilung in Submodelle

Wenn einzelne Komponenten eines Applikationssystems wie Dialoge oder Grammatiken wiederverwendet oder von mehreren Benutzern gleichzeitig verändert werden sollen, müssen diese deutlich voneinander getrennt sein und z. B. in eigenen Dateien gespeichert werden. Trotzdem müssen diese Teile direkt vom Wurzelknoten des Hauptmodells referenziert werden. Sonst gibt es Probleme wenn ausgehend von der Wurzel des Systemmodells (das in beliebigen Unterknoten diese externen Elemente referenziert) alle benutzten Elemente eindeutig erreicht werden sollen, was vor allem für die Validierung wichtig ist. Deshalb muss bei der Aufteilung in Submodelle auf eine direkte Einbindung aller eingebetteten Komponenten im Wurzelobjekt des Dialogsystems geachtet werden.

5.6 Fehlende Konzepte

Leider gibt es im sehr üppig ausgestatteten Eclipse Universum noch einige Lücken im Bereich der modellbasierten Entwicklung, die eine Umsetzung von wünschenswerten Eigenschaften leider unmöglich gemacht haben.

5.6.1 Modellmigration

Es gibt kein allgemeines und in die üblichen Werkzeuge integriertes Verfahren für eine automatische Migration der Modelle nach einer Änderung des Metamodells. Dadurch mussten die Dialogmodelle nach einer Änderung des Metamodells meist komplett neu erstellt werden.

5.6.2 Modellierung des Verhaltens

Die auf Basis von EMF entwickelten Metamodelle beschreiben nur die Eigenschaften der Modellelemente und nicht ihr Verhalten. Dieses kann nur indirekt über Validierungsregeln getestet werden und lässt sich nicht für eine Verhaltenssimulation nutzen. Es gibt aber Ansätze dafür im Rahmen verschiedener Eclipse Projekte.

5.6.3 Kopplung zwischen grafischer und textueller DSL

Die grafische und textuelle Darstellung des Dialogmodells mit Hilfe von speziellen Editoren, die aus dem Metamodell generiert werden, wird von Eclipse vorbildlich unterstützt, allerdings gibt es keine Möglichkeit, die beiden Darstellungen automatisch ineinander zu überführen. Für Änderungen an der grafischen Darstellung gibt es also keinen automatisch erzeugten Prozess, um die textuelle Darstellung entsprechend anzupassen. Da beide Darstellungsformen ihre spezifischen Vor- und Nachteile haben, wäre eine solche Kopplung der beiden Sprachen für das Metamodell ein Gewinn für das gesamte System. Z. B. könnte das Dialogmodell dann in seiner eigenen DSL gespeichert werden, was Versionierung und das Zusammenführen von verteilt bearbeiteten Modellen deutlich erleichtern würde.

6 Modellbasierte Dialogentwicklung

Für die Entwicklung einer dialoggesteuerten Applikation werden zwei Modelle benötigt:

1. ein Modell für den Sprachdialog (so abstrakt wie möglich)

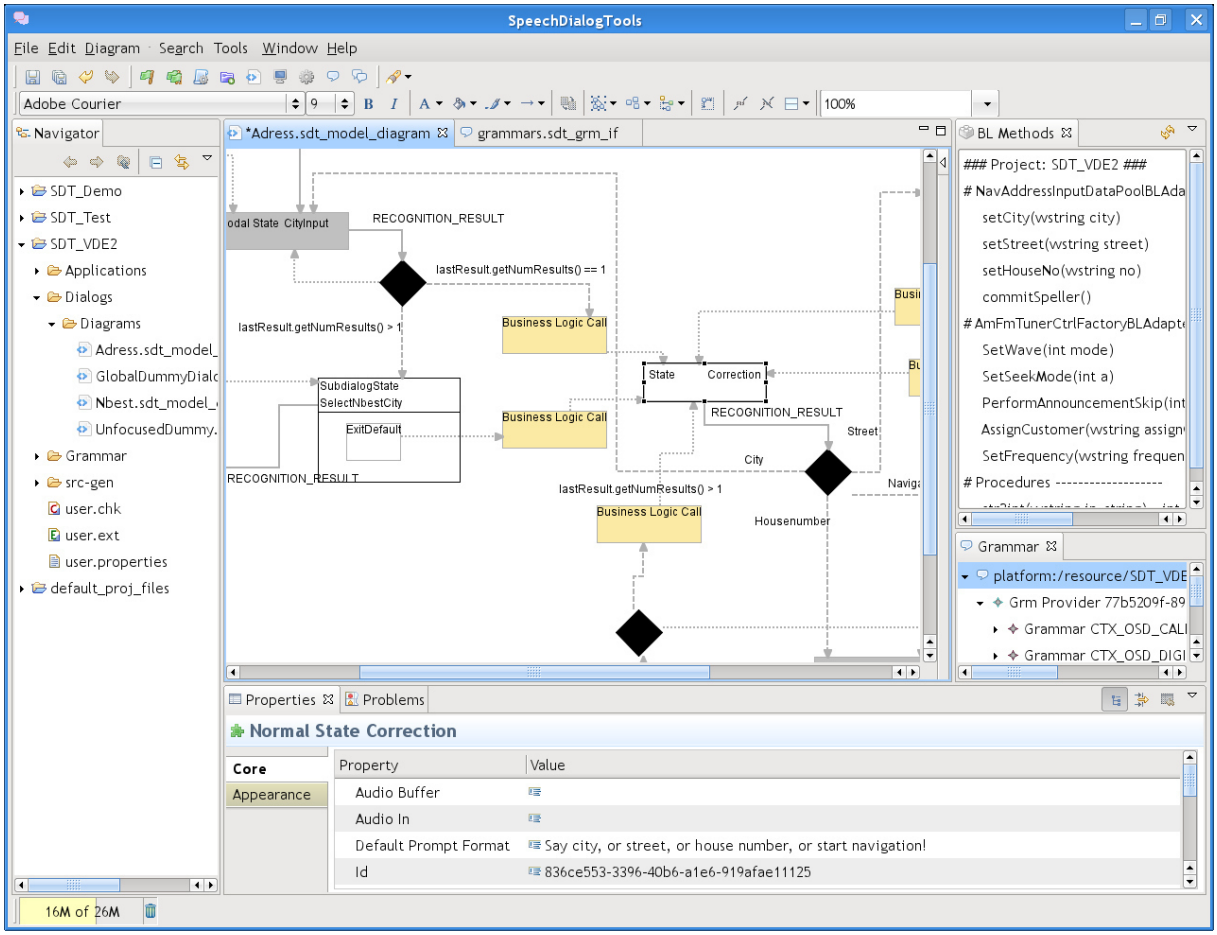

Abbildung 3 - SpeechDialogTools Editor mit Beispielmodell

2. ein Modell für die Zielplattform (konkret)

Zur Beschreibung dieser beiden Modelle, benutzt man ein Metamodell, das auch als domänenspezifische Sprache (DSL, Domain Specific Language) bezeichnet werden kann, da man mit seiner Hilfe ein Modell beschreibt. Dieses Metamodell beschreibt die Elemente, aus denen ein Dialog aufgebaut ist, und in welcher Beziehung sie zueinander stehen.

Mit Hilfe des Metamodells können nun Werkzeuge zur Unterstützung des Modellierungsprozesses gebaut werden, z. B. ein grafischer Dialogeditor (siehe Abb. 3) oder ein Editor für die aus dem Metamodell abgeleitete textuelle DSL.

7 Mustersprache für Dialogdesign

Auch wenn durch Verwendung der Komponenten des Eclipse Universums schon ein hoher Grad an Wiederverwendung bei der Entwicklung eines (grafischen) Editors für einen Sprachdialog erreicht ist, muss doch für eine neue Anforderung das Metamodell wieder von Grund auf neu entwickelt werden. Außer dem Rückgriff auf die Erfahrungen abgeschlossener Projekte bestehen kaum Möglichkeiten, konkrete Konzepte wiederzuverwenden. Dazu fehlt es an einer Sprache, um die Dialogkonzepte selbst zu beschreiben – es fehlt quasi eine DSL um die Modellelemente eines Dialogs und die konkrete Umsetzung für die verschiedenen Zielplattformen zu beschreiben.

Für diesen Zweck gibt es in anderen Bereichen sogenannte Mustersprachen (Pattern Languages, siehe [3, 4, 13]), die es ermöglichen, diese Konzepte zu beschreiben und auszutauschen.

Für das Design von grafischen Interfaces und hier besonders beim Modellieren von Aufgaben (Tasks) und (grafischen) Dialogen gibt es einige Ansätze für eine gemeinsame Sprache [11, 12, 14]. Für die Domäne der Modellierung von Sprachdialogen fehlt eine solche gemeinsame Mustersprache bisher.

Aus den Erfahrungen, die wir mit der modellbasierten Beschreibung und Entwicklung von Sprachdialogen für eingebettete Systeme gemacht und die wir für dieses Projekt im Abschnitt 5 beschrieben haben, wollen wir den Grundstein für eine solche Sprache legen und mit diesem Vortrag zur gemeinsamen Arbeit daran einladen.

Literatur

[1] BOOCH, G., J. RUMBAUGH und I. JACOBSON: *Unified Modeling Language User Guide*. Addison-Wesley Professional, 2005.

[2] BUDINSKY, F., E. MERKS und D. STEINBERG: *Eclipse Modeling Framework 2.0 (2nd Edition)*. Addison-Wesley Professional, 2008.

[3] C2WIKI: *Pattern Language*. `http://c2.com/cgi/wiki?PatternLanguage`.

[4] C2WIKI: *Portland Pattern Repository*. `http://c2.com/ppr/`.

[5] ECLIPSE FOUNDATION: *Legal Resources*. `http://www.eclipse.org/legal/`.

[6] ECLIPSE PROJECT: *Eclipse Modeling Project*. `http://eclipse.org/modeling/`.

[7] EFFTINGE, S., P. FRIESE und J. KÖHNLEIN: *Best Practices for Model-Driven Software Development*. `http://www.infoq.com/articles/model-driven-dev-best-practices`, June 2008.

[8] GRONBACK, R. C.: *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 2009.

[9] *openArchtitectureWare*. `http://www.openarchitectureware.org/`.

[10] PIETREK, G. und J. TROMPETER (Hrsg.): *Modellgetriebene Softwareentwicklung: MDA und MDSD in der Praxis*. Entwickler.press, Frankfurt am Main, 2007.

[11] SINNIG, D., P. FORBRIG und A. SEFFAH: *Patterns in Model-Based Development*. In: *INTERACT 03 Workshop*, 2003.

[12] WELIE, M. VAN: *Patterns in Interaction Design*. `http://welie.com/patterns/`.

[13] WIKIPEDIA: *Mustersprache*. `http://de.wikipedia.org/wiki/Mustersprache`.

[14] WURDEL, M., P. FORBRIG, T. RADHAKRISHNAN und D. SINNIG: *Patterns for Task- and Dialog-Modeling.*. In: JACKO, J. A. (Hrsg.): *HCI (1)*, Bd. 4550 d. Reihe *Lecture Notes in Computer Science*, S. 1226–1235. Springer, 2007.